Sharing a Sequential Program: Correctness and Concurrency Analysis

Vincent Gramoli [1] Petr Kuznetsov [2] Srivatsan Ravi [2]
[1] School of Information Technologies, University of Sydney
*
[2] TU Berlin/Telekom Innovation Laboratories †

Abstract

It seems to be generally accepted that designing correct and efficient concurrent software is a sophisticated task that can only be held by experts. A crucial challenge then is to convert sequential code produced by a "mainstream" programmer into concurrent one. Various synchronization techniques may be used for this, e.g., locks or transactional memory, but what does it mean for the resulting concurrent implementation to be correct? And which synchronization primitives provide more efficiency at the end?

In this paper, we introduce a correctness criterion for a concurrent "wrapper" of a sequential program, i.e., a transformation that enables the use of a sequential data structure in a concurrent system. Informally, we require the resulting concurrent implementation to be *locally sequential*: concurrent threads simply run the given sequential code and let the implementation worry about the potential conflicts. To make sense globally, the implementation should also be *linearizable* with respect to the *object type* of the data structure.

We then evaluate the performance of different concurrent implementations in terms of the sets of *schedules* (interleavings of steps of the sequential code) they *accept*. Intuitively, this captures the amount of concurrency that a given implementation can stand. This allowed us to analyze relative power of seemingly incomparable synchronization techniques, such as various forms of locking and transactional memory.

1 Introduction

Implementing a correct and efficient data structure shared by multiple users is believed to be a tedious and error-prone process. What if, instead, we use a concurrent "wrapper" of a *sequential* implementation of the data structure that allows every user to *locally* run its sequential code so that the resulting concurrent execution is *globally* correct.

One way to do this is to use *locks* so that critical parts of a sequential program are only accessed in an exclusive mode. An implementation that grabs a lock on the whole data structure before executing a sequential operation imposes a serial order of operations but neglects the benefits provided by the multiprocessing power of modern machines. Efficient fine-grained locking requires

*Part of this work was done when the author was with EPFL receiving funding from grant agreement N 248465, the S(o)OS project.

†The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement N 238639, ITN project TRANSFORM.

lots of intelligence, since it must be based on good understanding of which parts of the sequential code to protect at what time.

A more automated approach is to use transactional memory (TM) and treat each (sequential) operation as a speculative *transaction*. If the transaction commits, the corresponding operation returns the response computed based on the state the transaction witnessed. If the transaction aborts, the operation does not take effect. This approach promises to make use of the hardware concurrency at low intellectual cost. But does this simplicity bring an efficiency degradation?

Local serializability and LS-linearizability. We start with defining the very meaning of using a sequential data structure in a concurrent environment. One natural correctness requirement is that no thread ever reaches a state not anticipated by the sequential implementation. More precisely, we model an execution of a concurrent implementation as a sequence of invocations and responses of the *high-level* operations on the data structure (e.g., *Insert*, *Remove*, or *Contains* calls on a set object), events of the corresponding sequential implementation (e.g., reads and writes to the items of a sorted linked-list used to implement a set), plus accesses to synchronization primitives (e.g., transaction delimiters or acquisitions and releases of locks). Now we say that an execution is *locally serializable* if the sequence of sequential events corresponding to each high-level operation is consistent with *some* sequential execution. The condition is weaker than *serializability* [19] since it allows sequential executions corresponding to different high-level operations to be mutually inconsistent. In particular, there may not exist a single sequential execution on the given data structure that is consistent with all local executions. But it is good enough as long as we only want our concurrent data structure to locally appear sequential.

On the other hand, the implementation should "make sense" globally, given the *type* of our data structure. We require that the *high-level* history of every execution, i.e., the subsequence of high-level invocations and responses, is *linearizable* [3, 15]: each high-level operation should appear instantaneous at some point within its interval so that the high-level sequential semantics is respected.

The combination of local serializability and linearizability gives a correctness criterion which we call *LS-linearizability*, where LS stands for "locally serializable." Note that we can easily think of implementations that are linearizable but not LS-linearizable (do not look sequential locally), as well as locally serializable but not linearizable (do not make sense globally). Therefore, the two properties indeed complement each other. We also show that LS-linearizability inherits one important property of linearizability [14, 15]: it is *compositional*, a composition of LS-linearizable implementations is also LS-linearizable.

For example, consider the set type implemented sequentially using a sorted linked list (cf. Algorithm 1 in Appendix A). To check if the set contains a given value in a concurrent execution, we do not need to take a snapshot of the linked-list content. It is enough to make sure that each *two* consecutive elements of the list are read atomically. Indeed, the resulting LS-linearizable concurrent set implementation (described in detail in Appendix B) only requires an operation to hold locks on two consecutive list elements at a time (so called *hand-over-hand* locking [4, 14]). As a result, the *Contains*() operation may run concurrently with multiple *Insert*() operations, as long as they do not contend on the same consecutive pair of list elements.

Relative concurrency. The sorted linked-list example above suggests that we can evaluate the performance of an implementation via the "amount of concurrency" it allows for. More precisely, we can associate the implementation with the set of *schedules* (interleavings of steps of the sequential operations invoked by multiple users) it *accepts*, i.e., is able to process. Now we can

compare different concurrent implementations (or implementation classes) for a given sequential data structure (or a class of data structures) based on the sets of accepted schedules. It is particularly intriguing to compare the concurrency properties provided by various classes of TMs against locking techniques. Indeed, a TM interacts with its user application through a fixed transactional interface (*reads*, *writes*,*tryCommit* or *tryAbort*), and have no clue about the object's semantics, unlike lock-based implementations. On the other hand, TMs may "undo" their operations by aborting the corresponding transactions, which cannot be achieved with locks. The conditions under which a TM may (or have to) abort a transaction (e.g., captured via *progress* conditions [11]) are of crucial importance here, since, intuitively, they allow the TM to distinguish correct (observed in LS-linearizable histories) schedules from incorrect ones.

Results of concurrency analysis. We show first that fine-grained locking provides strictly more concurrency than a wide class of *conflict-resolving* TM-based implementations (resolving conflicts between concurrent transactions by forcefully aborting some of them) [12]. We describe a lock-based implementation that accepts every schedule with no conflicts, i.e., locking provides *at least as much* concurrency as conflict-resolving TMs. Moreover, we present a schedule that cannot be accepted by *any* strictly serializable TM (be it conflict-resolving or not), yet the schedule is accepted by a linked-list implementation using hand-over-hand locking [4, 14]. As a corollary, we derive that locking provides *strictly* more concurrency than *conflict-resolving* TMs. This is the price to pay for TMs being "oblivious" to the semantics of the object they implement, combined with the brute-force approach to conflict resolution.

However, we show that the price is not inherent for TM implementations that enforce stronger progress conditions. The use of multi-version concurrency control allows implementing *MV-permissive* [20] strictly serializable TMs in which *read-only* transactions always commit. Surprisingly, TM implementations in this class accept schedules which cannot be accepted by *any* lock-based implementation. Nevertheless, as handling multiple versions is known to come at a significant cost [20], it is hard to translate such concurrency improvements to performance gains. In response to this, we extend our result to *single-version* strictly serializable TMs that abort read-only transactions only if they witnesses two or more concurrent updates. We conjecture further that TMs with *relaxed* consistency guarantees, such as elastic transactions [5, 9], may be used to match the performance of locks for a large class of search data structures (queues, hash tables or trees).

At a large scale, we cannot definitively say that locks supersede transactions or vice versa. To determine the amount of concurrency provided by a synchronization technique, one should carefully specify its correctness and progress conditions, as well as on the type and sequential implementations of the given data structure. This paper gives a language in which relative concurrency of seemingly incomparable synchronization techniques can be properly analyzed.

Roadmap. Section 2 gives basics of our system model. Section 3 introduces LS-linearizability and defines our concurrency relations. Section 4 presents a comparative concurrency analysis of lock-based implementations and TM-based ones. We conclude by discussing related work and open questions in Sections 5 and 6. Omitted proofs can be found in the optional appendix.

2 Model

In this section, we define what we mean by *transforming* a sequential type implementation into a concurrent one, and describe two synchronization techniques: locks and transactional memory.

Sequential types and Implementations. An *object type* τ is specified by the tuple $(\Phi, \Gamma, Q, q_0, \delta)$ where Φ is a set of operations, Γ is a set of responses, Q is a set of states, $q_0 \in Q$ is an initial state and $\delta \subseteq Q \times \Phi \times Q \times \Gamma$ is a transition relation that determines, for each state, and each operation, the set of possible resulting states and produced responses. Hence, $(q, \pi, q', r) \in \delta$ implies that when an operation $\pi \in \Phi$ is applied on an object of type τ in state q, the object moves to state q' and returns a response r. We assume that δ is *total*, i.e., for every $q \in Q$, $\pi \in \Phi$, there exist $q' \in Q$ and $r \in \Gamma$ such that $(q, \pi, q', r) \in \delta$. We assume that every type $\tau = (\Phi, \Gamma, Q, q_0, \delta)$ is *computable*, i.e., there exists a Turing machine that, for each input (q, π), $q \in Q$, $\pi \in \Phi$, computes a pair (q', r) such that $(q, \pi, q', r) \in \delta$. Hence, we can construct a *sequential implementation* of type τ as follows. For each operation π, we define a deterministic procedure $\mathcal{P}^\pi$ that performs *reads* and *writes* on a collection of objects $X_1, \ldots, X_m$, $m \in \mathbb{N}$. The value returned by $\mathcal{P}^\pi$ is treated the response to π returned by the implemented object.

Concurrent implementations. We focus on the problem of turning a given sequential implementation I_S of type τ into a *concurrent* one, shared by n *processes* (users) $p_1, \ldots, p_n$ ($n \in \mathbb{N}$). Each sequential procedure $\mathcal{P}^\pi$ is converted into a concurrent one by inserting synchronization constructs (such as locks or delimiters of software transactions) between reads and writes performed by $\mathcal{P}^\pi$. Intuitively, the resulting concurrent implementation must locally (to each particular operation) look like the sequential implementation I_S, while being globally correct with respect to τ (a precise definition of our correctness criterion is given in Section 3).

Executions. An *execution* of a concurrent implementation is thus a sequence of invocations and responses of high-level operations, (atomic) read and write events and *synchronization* events (e.g., lock acquisitions and releases or transaction delimiters). We assume that executions are *well-formed*: no process invokes a new high-level operation before the previous one returns or takes steps outside its operation's interval. In this paper, we primarily focus on two synchronization techniques: locks and transactional memory (TM), described in detail below.

Exported histories. Informally, a *history exported by an execution E* is the sequence of invocations and responses of high-level operations and read-write events that *take effect* in E. The exact definition depends on the type of synchronization primitives used in the execution, and should be defined separately in each case (we define it for locks and TMs below). By convention, every execution of a sequential implementation I_S is already a history. Histories H and H' are *equivalent* if, for every process p_i, $H|p_i{=}H'|p_i$.

A *high-level history* $\tilde{H}$ of an execution E is the sequence of invocations and responses on high-level objects in E. A high-level operation π *precedes* another high-level operation π' in $\tilde{H}$, denoted $\pi \rightarrow_{\tilde{H}} \pi'$, if the response of π occurs before the invocation of π'. Two high-level operations are *concurrent* if neither precedes the other. A high-level history is *sequential* if it has no concurrent operations. $\tilde{H}$ is *complete* if every invocation has a corresponding response. A history H is *complete* if the corresponding high-level history $\tilde{H}$ is complete.

A complete high-level history $\tilde{H}$ is *linearizable* with respect to an object type τ if it is possible to permute events in $\tilde{H}$ to obtain a sequential high-level history S such that (1) $\rightarrow_{\tilde{H}} \subseteq \rightarrow_S$ and (2) S is consistent with type τ. Now an arbitrary history $\tilde{H}$ is linearizable if it can be *completed* (by adding matching responses to a subset of incomplete operations in $\tilde{H}$ and removing the rest) to a linearizable high-level history [3, 15].

Locks. A *lock* provides exclusive access to an object X and is accessed through atomic operations *acquire*$^S(X)$ (*shared mode*), *acquire*$^E(X)$ (*exclusive mode*) and *release(X)*. If no process holds the

lock on some object X in shared or exclusive mode, then *acquire*$^E(X)$ returns *true*; if no process holds the lock on some object X in exclusive mode, then *acquire*$^S(X)$ returns *true*.

Given a sequential data structure, a corresponding concurrent lock-based one is obtained by inserting *acquire* and *release* events between the read-write events of each sequential operation. We assume a *dynamic* locking scheme: a process may only acquire a lock on X if X is the next object it is going to access. Also, a process releases all locks it acquired before returning the result of the operation. The history exported by an execution E of a lock-based implementation is the sequence of events obtained from E by removing all synchronization (acquire/release) events.

Transactional memory. A transactional memory (*TM*) provides access to a collection of shared objects (*t-objects*) via *atomic transactions*. A transaction T_k is a sequence of *reads* and *writes* on t-objects that begins with *start-txn* and ends with *tryCommit* or *tryAbort*, where *tryCommit* returns either *commit* (the transaction takes effect) or *abort* (the transaction does not take effect), and *tryAbort* aborts the transaction. Each transaction T_k and all its events are associated with a unique identifier k. Transactions T_i, T_j *conflict* in an execution E on a t-object X if there exists a prefix of E in which T_i and T_j are incomplete and access X, and at least one of these accesses is a *write*. A transaction T_k is *t-complete* in E if it ends with *tryCommit* or *tryAbort*. We say that E is *t-complete* if it contains only t-complete transactions. For T_k, T_m in E, we say that T_k *precedes* T_m in the *real-time order* in E, if T_k is committed or aborted and the last event of T_k precedes the first event of T_m in E. Two transactions are *concurrent* if neither precedes the other. E is *t-sequential* if no two transactions are concurrent in E. We say that a t-complete t-sequential history E is *legal* if for every t-object X, every read of X in E returns the latest written value of X. Let $\bar{E}$ denotes the subsequence of E consisting of the events of commited transactions. A t-complete execution E is *strictly serializable* if there exists a legal complete t-sequential history S such that (1) $\bar{E}$ and S are equivalent and (2) S respects the real-time order of transactions. Now we say that a TM-based execution is strictly serializable if we can extend it to produce a t-complete strictly serializable execution. A TM implementation M is strictly serializable if every execution of M is strictly serializable.

Note that we make a simplifying assumption that all TM operations appear atomic, which applies to most existing TMs. In terms of liveness, we assume that these operations are starvation-free: as long as every process is correct, every tm-operation eventually returns. In this paper, we primarily focus on *strictly serializable* TMs (formal definition in Appendix C).

Given a sequential data structure, a corresponding TM-based concurrent one puts each sequential operation within a transaction, i.e., every operation π on it is implemented as: *start-txn*; $\mathcal{P}^\pi$; *tryCommit*, where $\mathcal{P}^\pi$ is the sequential implementation of π. If *tryCommit* returns *commit*, then the result of $\mathcal{P}^\pi$ is returned to the user. The history exported by an execution E of a TM-based implementation is the subsequence of E obtained by removing all synchronization events and all events related to *aborted* or *incomplete* transactions.

3 LS-linearizability

Let H be a history, and let π be a high-level operation in H. Then $H|\pi$ denotes the subsequence of H consisting of the events of π. Let I_S be a sequential implementation of an object of type τ.

Definition 1 *A history H is* locally serializable with respect to I_S *if for all high-level operations π in H, there exists $S \in \Sigma$, where Σ is the set of histories of I_S, such that $H|\pi = S|\pi$.*

Note that local serializability stipulates that the execution is witnessed sequential by every high-level operation in isolation. Two different operations (even when invoked by the same process) are not required to witness mutually consistent sequential executions.

Definition 2 *A history H is* LS-linearizable with respect to (I_S, τ) *if (1) H is locally serializable with respect to I_S, and (2) the corresponding high-level history $\tilde{H}$ is* linearizable with respect to τ.

A correctness property is *compositional* [14, 15] if, informally, a composition of correct object implementations is also correct. LS-linearizability is compositional (Appendix **??**). It is not *non-blocking* [14, 15], however, in the sense that an incomplete operation may not be able to complete on its own, which may be seen as a feature inherent to local serializability.

We define the composition of two distinct object types τ_1 and τ_2 as a type $\tau_1 \times \tau_2 = (\Phi, \Gamma, Q, q_0, \delta)$ as follows: $\Phi = \Phi_1 \cup \Phi_2$, $\Gamma = \Gamma_1 \cup \Gamma_2$,[1] $Q = Q_1 \times Q_2$, $q_0 = (q_{01}, q_{02})$, and $\delta \subseteq Q \times \Phi \times Q \times \Gamma$ is such that $((q_1, q_2), \pi, (q_1' q_2'), r) \in \delta$ if and only if $\pi \in \Phi_i$ $(i = 1, 2)$ and $(q_i, \pi, q_i', r) \in \delta_i$.

Every sequential implementation I_S of an object $O_1 \times O_2$ of a composed type $\tau_1 \times \tau_2$ naturally induces two sequential implementations I_{S1} and I_{S2} of objects O_1 and O_2, respectively. Now a correctness criterion Ψ is *compositional* if for every history H on an object composition $O_1 \times O_2$, if Ψ holds for $H|O_i$ with respect to I_{Si}, for $i = 1, 2$, then Ψ holds for H with respect to $I_S = I_{S1} \times I_{S2}$. Here, $H|O_i$ denotes the subsequence of H consisting of steps on O_i.

Theorem 3 *LS-linearizability is compositional.*

Proof. Let H, a history on $O_1 \times O_2$, be LS-linearizable with respect to I_S. Let each $H|O_i$, $i \in \{1, 2\}$, be LS-linearizable with respect to I_{Si}. Without loss of generality, we assume that H is complete (if H is incomplete, we consider any completion of it containing LS-linearizable completions of $H|O_1$ and $H|O_1$).

Let $\tilde{H}$ be the history of H. By the assumption, $\tilde{H}|O_1$ and $\tilde{H}|O_2$ are linearizable with respect to τ_1 and τ_2, respectively. Since linearizability is compositional [14, 15], $\tilde{H}$ is linearizable with respect to $\tau_1 \times \tau_2$.

Now let, for each operation π, S_π^1 and S_π^2 be any two sequential histories of I_{S1} and I_{S2} such that $H|\pi|O_j = S_\pi^j|\pi$, for $j = 1, 2$ (since $H|O_1$ and $H|O_2$ are LS-linearizable such histories exist). We construct a sequential history S_π by interleaving events of S_π^1 and S_π^2 so that $S_\pi|O_j = S_\pi^j$, $j \in \{1, 2\}$. Since each S_π^j acts on a distinct component O_j of $O_1 \times O_2$, every such S_π is a sequential history of I_S. We pick one S_π that respect the local history $H|\pi$, which is possible, since $H|\pi$ is consistent with both $S_1|\pi$ and $S_2|\pi$.

Thus, for each π, we obtain a history of I_S that agrees with $H|\pi$. Moreover, the high-level history of H is linearizable. Thus, H is LS-linearizable. $\square$

It is easy to see that LS (locally serializable) linearizability is distinct from both serializability [19] and linearizability [15]. For example, the classical queue implementation by Michael and Scott [17] is linearizable, but allows threads to witness states that can only arise in concurrent executions (e.g., when underlying CAS operations fail).

Also, consider the history of a sorted linked-list implementation depicted in Figure 2. Here we assume that the initial state of the set is $\{1, 3, 4, 5\}$, object X_i $(i = 1, \ldots, 6)$ is used for storing value i, and operation *Contains*() returns *true*. The high-level history is linearizable with respect to the

[1] Here we treat each τ_i as a distinct type by adding index i to all elements of Φ_i, Γ_i, and Q_i.

$\mathsf{Integer\ Set}$ type (*Contains*() is linearized after the two *Insert*() operations). But since *Contains*() observes an old value of X_1 (present before *Insert*(2)) and a new value of X_5 (written by *Insert*(6)), and *Insert*(6) reads the value of X_1 written by *Insert*(2), there is no way to find a global serialization for the three operations, i.e., the history is not serializable. But, as we show in Section 4, there exists an LS-linearizable $\mathsf{Integer\ Set}$ implementation based on hand-over-hand locking that exports this history.

Preliminary observations. *Two-phase locking* (*2PL*) [23] is a lock-based implementation technique in which every *read* of an object in the course of an operation involves acquiring a *shared* lock and every *write* involves acquiring an *exclusive* lock, and all acquired locks are released before returning the result of the operation. Thus, every sequential procedure of a sequential implementation I_S is converted into a concurrent one by inserting $acquire^S(X)$ before $read(X)$ and $acquire^E(X)$ before $write(X)$. The corresponding *release* events are inserted at the end of the procedure. Let I_S^{2PL} denote a concurrent implementation of a sequential data structure I_S based on *2PL*. Since *2PL* guarantees strict serializability [23], we immediately obtain:

Observation 4 *For any sequential implementation I_S of a type τ, I_S^{2PL} is LS-linearizable with respect to (I_S, τ).*

Similarly, if we transform a sequential implementation I_S of type τ into a concurrent one using $\mathcal{M}$, by treating every operation of I as a transaction, we obtain a strictly serializable and linearizable implementation $I_S^{\mathcal{M}}$, which implies:

Observation 5 *For any strictly serializable TM implementation $\mathcal{M}$, and any sequential implementation I_S of a type τ, $I_S^{\mathcal{M}}$ is LS-linearizable with respect to (I_S, τ).*

Consider the sequential implementation LL of the $\mathsf{Integer\ Set}$ type (or *set* for brevity) described in Algorithm 1 in Appendix A. The type *set* maintains operations *Insert*, *Remove* and *Contains* with the usual semantics. We describe a concurrent implementation, I^{HOH}, derived from LL based on *hand-over-hand* locking [4] (*HOH*) (Algorithm 2 in Appendix B).

Theorem 6 *I^{HOH} is LS-linearizable with respect to (LL, set).*

3.1 Concurrency relations

A *schedule* is an equivalence class of histories that agree on the order of events but possibly not on read values or high-level responses. Intuitively, a schedule describes the order in which high-level operations, and sequential reads and writes are invoked by the user. We say that an implementation I *accepts* a schedule S if there exists an execution of I which exports a history exhibiting the order of S. Given a concurrent implementation I, let $\mathcal{S}(I)$ denote the set of schedules accepted by I. Intuitively, $\mathcal{S}(I)$ reflects the "amount of concurrency" provided by I.

A *synchronization technique* is a set of concurrent implementations. Given a sequential implementation I_S of an object type τ and a synchronization technique $\mathcal{A}$, let $\mathcal{T}_{\mathcal{A}}(I, \tau)$ denote the set of LS-linearizable (with respect to (I_S, τ)) implementations in $\mathcal{A}$. We say that a synchronization technique $\mathcal{A}$ provides *less concurrency* than a synchronization technique $\mathcal{B}$ *with respect to* (I_S, τ), and we write $\mathcal{A} \preceq_{(I_S,\tau)} \mathcal{B}$, *iff* $\forall I \in \mathcal{T}_{\mathcal{A}}(I_S, \tau), \exists I' \in \mathcal{T}_{\mathcal{B}}(I_S, \tau)$, $\mathcal{S}(I) \subseteq \mathcal{S}(I')$.

We say that $\mathcal{A}$ provides *strictly less concurrency than $\mathcal{B}$ with respect to* (I_S, τ), and we write $\mathcal{A} \prec_{(I_S,\tau)} \mathcal{B}$, *iff* $(\mathcal{A} \preceq_{(I_S,\tau)} \mathcal{B}) \wedge (\mathcal{B} \not\preceq_{(I_S,\tau)} \mathcal{A})$.

If $\mathcal{A} \preceq_{(I_S,\tau)} \mathcal{B}$ for all (I_S, τ), then we say that $\mathcal{A}$ *provides less concurrency than* $\mathcal{B}$ and write $\mathcal{A} \preceq \mathcal{B}$. Similarly, we write $\mathcal{A} \prec \mathcal{B}$, *iff* $(\mathcal{A} \preceq \mathcal{B}) \wedge (\mathcal{B} \not\preceq \mathcal{A})$.

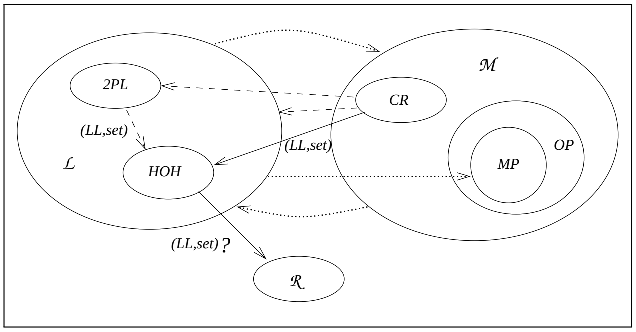


Figure 1: Summary of comparative analysis of synchronization techniques

$\mathcal{L}$	dynamic fine-grained locking
2PL	two-phase locking
HOH	hand-over-hand locking
$\mathcal{M}$	strictly-serializable TMs
CR	conflict-resolving TMs
MP	MV-permissive TMs
OP	1-progressive TMs
$\mathcal{R}$	relaxed TM (conjectured)
- - ->	$\preceq$
——>	$\prec$
$\cdots$>	$\not\preceq$
(LL, set)	with respect to (LL, set)

Table 1: Notations

4 Locks vs. Transactional memory: Concurrency Analysis

In this section, we use our concurrency relations to compare two synchronization techniques: locking (denoted $\mathcal{L}$) and strictly serializable transactional memory (denoted $\mathcal{M}$). We also consider three subclasses of $\mathcal{M}$: *conflict-resolving* implementations (*CR*), *MV-permissive* implementations (*MP*) and 1-*progressive* implementations (*OP*), and two subclasses of $\mathcal{L}$: *two-phase locking* (*2PL*) and *hand-over-hand locking* (*HOH*). A summary of our results is presented in Figure 1.

4.1 Conflict-resolving TMs

We say that an TM implementation is *conflict-resolving* if it exports no history in which two committed transactions experience a *conflict*. Intuitively, in conflict-resolving TMs, such as [12], a transaction that witnesses a read-write conflict at some point during its execution is forcefully aborted or delayed. The class of implementations based on conflict-resolving TMs is denoted by *CR*. *2PL* ensures that

Theorem 7 *CR* $\preceq$ *2PL*.

Proof sketch. Let $I \in CR$. Let schedule S be accepted by I, and let H be the corresponding history be exported by I. Recall that H only contains events from committed transactions. Suppose, by contradiction, that S is not accepted by I^{2PL}, i.e., some $acquire^S$ or $acquire^E$ invoked in H is unsuccessful. But then H has a prefix in which two high-level operations π and π' (1) are *not complete* and (2) both access the same object and at least one of these accesses is a *write*. Hence, the execution of I that exported H contains two conflicting committed transactions—a contradiction with the assumption that $I \in CR$. Thus, $\mathcal{S}(I) \subseteq \mathcal{S}(2PL)$. $\square$

Now we show that there exist schedules that are accepted by some implementations in $\mathcal{T}_{\mathcal{L}}(LL, set)$ (in our case I^{HOH}, the *HOH* implementation of a linked list-based set) but not by *any* implementation in $\mathcal{T}_{TM}(LL, set)$. Specifically, consider the schedule depicted in Figure 2. We observe that no two consecutive reads performed by the *Contains*(6) operation witness a conflicting write and, thus, I^{HOH} accepts the schedule. On the other hand, there is no way to serialize the three operations respecting the real-time order for *Insert*(2) and *Insert*(6) and making sure that the execution of *Contains*(6) appears sequential. Thus:

Theorem 8 *HOH* $\not\preceq_{(LL,set)} \mathcal{M}$.

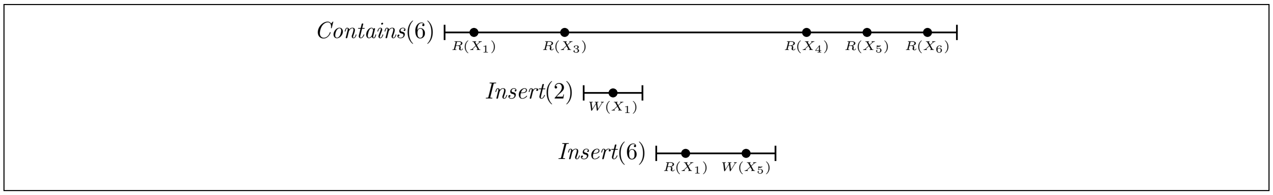


Figure 2: A schedule S_0 accepted by I^{HOH} but rejected by $I^{\mathcal{M}}$ (we only show read-write events that matter); the initial state of the list is $\{1, 3, 4, 5\}$

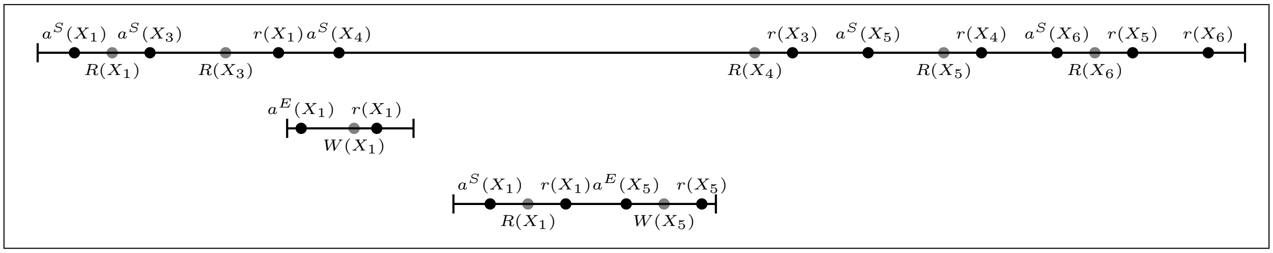


Figure 3: An execution of I^{HOH} that accepts S_0; a^S, a^E denote lock acquisition in shared, exclusive modes, r denotes the release event.

Proof. Consider processes p_1, p_2 and p_3 that execute operations *Contains*(6), *Insert*(2) and *Insert*(6) respectively. We show that the schedule S_0 in Figure 2 is accepted by I^{HOH}. Indeed, Figure 3 presents an execution of I^{HOH} that exports S_0. It is easy to see that the *acquire*^{S} on X_1, X_3, X_4 is successful since there are no concurrent writes performed on these objects in this execution. It is thus enough to verify that p_1 does not hold the shared lock on X_1 (resp., X_5) at the moment p_2 (resp., p_3) tries to acquire an exclusive lock on it. Process p_2 acquires the exclusive lock on X_1 to insert 2 after p_1 has released the lock on X_1. Process p_3 invokes *acquire*^{S} on X_1, X_2, X_3, X_4 and returns successfully (since *Insert*(2) precedes *Insert*(6) in real-time) and finally invokes *acquire*^{E} on X_5. Note that, at this point in the execution, p_1 does not hold the shared lock on X_5. Hence, the *acquire*^{E} on X_5 is successful. After the release of the lock on X_5 by p_3, process p_1 successfully acquires the shared lock on X_5 and X_6. Since $\{1, 3, 4, 5\}$ is the initial state of the list, *Contains*(6) is linearized after *Insert*(6) and returns *true* in H.

Suppose now that $S_0 \in \mathcal{S}(I^M)$, where M is a strictly serializable TM, and let H' be the corresponding history. Let transactions T_1, T_2 and T_3 encapsulate *Contains*(6), *Insert*(2) and *Insert*(6), respectively, in H'. Note that T_2 must precede T_3 in any serialization (real-time order). Since T_1 reads X_1 before T_2 updates it, T_1 must be serialized before T_2. Thus, to produce a legal serialization of reads and writes, T_1 must read the initial value of X_5, where $X_5.next$ points to the tail of the list (according to our sequential implementation I_S.

But then, according to I_S, *Contains*(6) cannot access X_6 in H', i.e., H' is not locally serializable (with respect to *Contains*(6))—a contradiction. $\square$

Corollary 9 $\mathcal{L} \not\preceq \mathcal{M}$.

Theorem 7 and Corollary 9 imply:

Corollary 10 $CR \prec \mathcal{L}$.

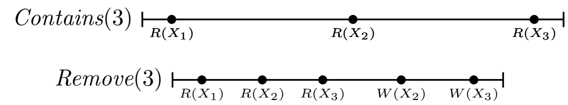


Figure 4: A schedule accepted by a *MV-permissive* TM implementation, but rejected by any lock-based implementation; the initial state of the list is $\{1, 2, 3\}$

4.2 MV-permissive TMs

An *MV-permissive* [20] TM allows a transaction to abort only if it is an updating transaction that conflicts with another updating transaction (In particular, no read-only transaction can abort). We denote by *MP* the class of implementations based on MV-permissive TMs.

Consider the schedule of an implementation of (LL, set) depicted in Figure 4. It is easy to see that the schedule does not contain update-update conflicts, so an MV-permissive TM must accept the schedule (enforcing $R(X_2)$ and $R(X_3)$ to return the initial values). The resulting history (assuming *Contains*(3) returns *true*) is linearizable and locally serializable with respect to (LL, set).

In contrast, for any implementation in $\mathcal{T}_{\mathcal{L}}(LL, set)$, $R(X_3)$ performed by *Contains*(3) must return the value written in $W(X_3)$ performed by *Remove*(3) (otherwise the execution is not serializable). But $W(X_3)$ marks X_3 as removed from the list, which can never be observed in a sequential execution of *LL*. Thus, the execution of *Contains*(3) is not locally serializable.

Theorem 11 $MP \not\preceq_{(LL,set)} \mathcal{L}$.

Proof. Consider a implementation in $\mathcal{T}_{\mathcal{L}}(LL, set)$ in which processes p_1 and p_2 execute operations *Contains*(3) and *Remove*(3), respectively. Process p_2 performs a *write* of X_3 that marks the node to be removed from the list and then p_1 performs the *read* of node X_3(which has been deallocated from the list). But the history exported by any such execution is incorrect. Thus, there does not exist a lock-based implementation $I^{\mathcal{L}}$ such that $\{S_1\} \in \mathcal{S}(I^{\mathcal{L}})$.

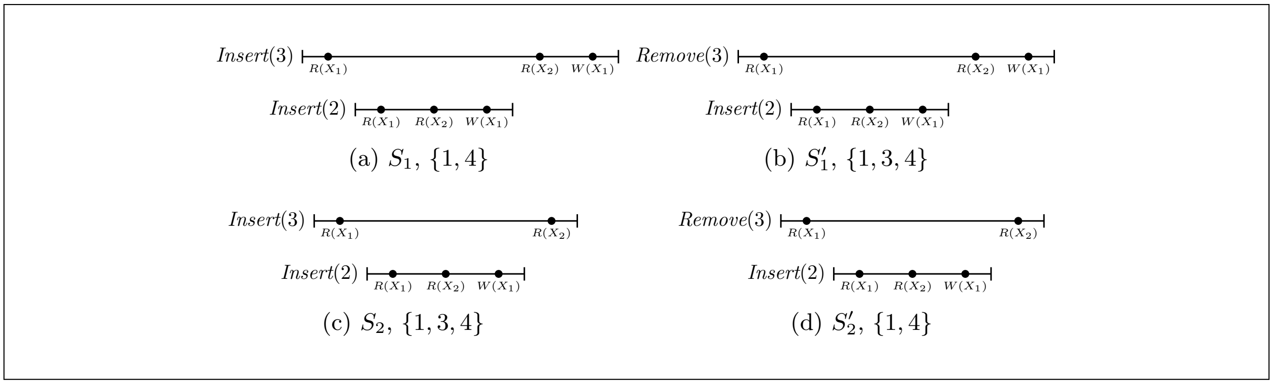
Let I^{MP} denote a TM implementation that belongs to class *MP*. Let E_1 denote an execution of I^{MP} in which transactions T_1, T_2 encapsulate *Contains*(3) and *Remove*(3) respectively. Since T_1 is read-only and T_2 only conflicts with T_2, both transactions must commit in E_1, and T_1, T_2 is the only serialization (here $R(X_2$ and $R(X_3)$ return the initial values in T_1). Also, the serialization is matched by a sequential execution of *LL* in which *Contains*(3) is followed by *Remove*(3). Thus,I^{MP} is indeed an LS-linearizable implementation of (LL, set) that accepts S_1. $\square$

Corollary 12 $\mathcal{M} \not\preceq \mathcal{L}$.

4.3 Single-version 1-progressive TMs

Our reasoning around the schedule in Figure 4 assumed MV-permissive TMs that may have to maintain exponentially many versions of transactional objects [20]. We now describe an alternative progress condition, called 1-*progressiveness*, that allows for *single-version* implementations. We further show that *OP*, the class of implementations based on 1-progressive TMs, is not superseded by $\mathcal{L}$, i.e., accept schedules that cannot be accepted by any lock-based implementation.

Formally, a TM implementation is 1-*progressive* if (1) an updating transaction aborts only if it conflicts with another updating and (2) a read-only transaction aborts only if it conflicts with



(a) S_1, $\{1,4\}$ (b) S_1', $\{1,3,4\}$

(c) S_2, $\{1,3,4\}$ (d) S_2', $\{1,4\}$

Figure 5: Schedules S_1, S_1', S_2, S_2', each schedule associated with its initial state

another transaction over two or more t-objects. In Appendix C, we present a *single-version* 1-progressive serializable TM implementation. Note that since 1-progressiveness is strictly weaker than MV-permissiveness, this does not contradict the impossibility result in [20].

Consider schedules S_1 and S_1' depicted in Figures 5a and 5b. In S_1, two concurrent processes p_1 and p_2 perform *Insert*(3) and *Insert*(2), respectively, and the initial state of the linked list is $\{1,4\}$, where 1 is stored in object X_1, 4 is stored in object X_2 and the *next* field of X_1 points to X_2. S_1' is similar to S_1 except that the initial state is $\{1,3,4\}$ now, p_1 performs *Remove*(3), X_1 stores 1 and X_2 stores 3. We observe first that neither S_1 nor S_1' can be accepted by a lock-based implementation of (LL, set).

Lemma 13 *$\forall I \in \mathcal{T}_{\mathcal{L}}(LL, set)$: $S_1 \notin \mathcal{S}(I)$ and $S_1' \notin \mathcal{S}(I)$.*

Proof. Suppose, by contradiction, that S_1 is accepted by a lock-based LS-linearizable linked-list implementation I. Let E be the execution of I in which S_1 is exported. At the end of E, p_1 overwrites the *next* field of X_1 that p_2 had previously set to point to node X_2 that stores value 2. Now X_1 points to a new object that stores value 3 resulting in a "lost-update problem". Indeed, consider an extension of E in which p_1 executes *Contains*(2) and returns. But, since the update of p_2 has been overwritten by p_1, the *Contains* operation may only return *false*. The resulting execution of I is not LS-linearizable (a previously inserted element 2 is not detected)—a contradiction.

Similarly, suppose that S_1' is accepted by some I and let E be the execution of I that exports S_1'. By the algorithm, p_1 writes to X_1 making sure that the *next* field of X_1 points to the object storing 4, and, thus, overwriting the update of p_2. Again, a subsequent *Contains*(2) would return *false*, violating LS-linearizability. □

Now consider schedules S_2 and S_2' depicted in Figures 5c and 5d. S_2 is like S_1, except that the initial state now is $\{1,3,4\}$ and, thus, *Insert*(3) performed by p_1 does not write to any object (the element is already in the set). Similarly, S_2' is like S_1', except that the initial state now is $\{1,4\}$ and, thus, *Remove*(3) performed by p_1 does not write to any object (the element is not in the set). It is easy to see that both schedules correspond to LS-linearizable histories. Note that p_2 is not able to distinguish executions exporting S_1 and S_2': in both executions it reads values 1 in X_1 and 4 in X_2. Similarly, p_2 cannot distinguish executions exporting S_1' and S_2. We observe also that p_1 cannot distinguish S_1 and S_2 (S_1' and S_2', respectively) up to the point when it executes $R(X_2)$. Therefore, intuitively, no lock-based implementation of (LL, set) can accept *both* S_2 and S_2'.

Lemma 14 *$\forall I \in \mathcal{T}_{\mathcal{L}}(LL, set)$: $\{S_2, S_2'\} \not\subseteq \mathcal{S}(I)$.*

Proof. Let I be any lock-based LS-linearizable linked-list implementation and suppose that I accepts both S_2 and S_2'. Let E and E' be the corresponding executions. We establish a contradiction by showing that I then also accepts S_1 or S_1'.

Since, by Lemma 13, S_1 cannot be accepted, and p_2 cannot distinguish S_2' and S_1, p_1 must use a different locking scheme when executing *Remove*(3) and *Insert*(3). In particular, in *Insert*(3) (schedules S_1 and S_2), p_1 must grab a lock on X_1 just before executing $R(X_1)$. (Recall that we assume the dynamic locking scheme, so p_1 can only acquire a lock on X_1 at this point.)

But since S_2 is accepted, p_2 does not grab a conflicting lock on X_1 before it distinguishes S_2 from S_1 (i.e., it has completed $R(X_2)$ and is about to execute $W(X_2)$). Thus, in S_1, p_1 must grab a conflicting lock on X_1 *just before* executing $W(X_1)$. Now we consider two cases:

- In S_1, p_2 acquires a shared lock on X_1 before executing $W(X_1)$. Thus, p_1 acquires an exclusive lock on X_1 before $R(X_1)$ in executing *Insert*(3) (in particular, in S_2).

  Since S_2 is accepted, p_2 holds no lock on X_1 in executing *Insert*(2) in E. But p_2 cannot distinguish S_2 and S_1'. Thus, in S_1', p_2 observes no conflicts while executing *Insert*(2) and completes the operation. On the other hand, when p_1 executes $R(X_2)$ and $W(X_1)$, p_2 has already terminated its operation and released all the locks. Thus, S_1' is also accepted by I—a contradiction with Lemma 13.

- In S_1 (and, thus, in S_2'), p_2 acquires an exclusive lock on X_1 before executing $W(X_1)$.

  Since S_2' is accepted, p_1 acquires no locks before $R(X_1)$ in S_2'. But p_1 cannot distinguish S_2' and S_1' before it executes $R(X_2)$. Thus, in S_1', p_2 observes no conflicts while executing *Insert*(2) and completes the operation. On the other hand, when p_1 executes $R(X_2)$ and $W(X_1)$, p_2 has already terminated its operation and released all the locks. Thus, S_1' is also accepted by I—a contradiction with Lemma 13.

Thus, I does not accept one of the schedules in $\{S_2, S_2'\}$. □

On the other hand, any implementation based on a 1-progressive TM (e.g., our single-version TM in Appendix C) ensures:

Lemma 15 $\forall I \in \mathcal{T}_{OP}(LL, set)$, $\{S_2, S_2'\} \subseteq \mathcal{S}(I)$.

Proof. Let E denote an execution of I in which transactions T_1 and T_2 encapsulate *Insert*(3) and *Insert*(2), respectively, interleaved as in schedule S_2. T_2 is an updating transaction that does not observe a conflict with another updating transaction in E. Also, T_1 is a read-only transaction that is allowed to abort only when it conflicts with another transaction over at least two t-objects. Thus, by 1-progressiveness, both T_1 and T_2 must commit in E.

The proof for S_2' is analogous to the proof for S_2. □

Lemmas 14 and 15 together with our implementation in Appendix C imply:

Theorem 16 *There exists $I^{OP} \in \mathcal{T}_{OP}(LL, set)$ based on 1-progressive single-version TM that cannot be superseded by locks, i.e., for any $I \in \mathcal{T}_{\mathcal{L}}(LL, set)$, $\mathcal{S}(I^{OP}) \nsubseteq \mathcal{S}(I)$.*

5 Related work

Our correctness criterion (LS-linearizability) not only requires the high-level history to be linearizable, but also put restrictions on the base object accesses, as in strengthened versions of linearizability [2, 7], but differs in that it expects the execution to look sequential with respect to a given sequential implementation (local serializability). In fact, local serializability can complement any of the restricted linearizability criteria in [2, 7].

The idea of measuring the concurrency properties of LS-linearizable implementations via the set of accepted schedules generalizes the notion of the *input acceptance* metric proposed to compare specific classes of TMs [10]. Our concurrency metric, however, is not restricted to the TM context and can be applied to any synchronization technique.

Empirical studies [6,21,22] suggested that transactions can be as efficient as fine-grained locking. Our results exhibit a much more involved picture of relative concurrency properties of TMs and locks, where TM's progress conditions is an important factor. Conflict-resolving TMs are shown to be superseded by locks. On the other hand, MV-permissive and 1-progressive TMs accept schedules that cannot be accepted by any lock-based implementation.

Locks and TMs were compared in the context of polymorphic and monomorphic transactions [8], with respect to a single schedule of a specific implementation, making it difficult to draw general conclusions. In [9], the programming interface of locks was argued to provide more expressiveness than transactions in terms of programming ability.

6 Concluding remarks

In this paper, we focused on implementations that allow for running sequential code in concurrent environments. We defined what it means for such implementations to be correct via a combination of linearizability and local serializability. Then we analyzed relative amount of concurrency properties of LS-linearizable implementations using fine-grained locking and TMs (cf. Figure 1).

This work has two important implications. First, it gives a language to reason about the "best" synchronization techniques for a given data structure. Second, if we are restricted to use just one synchronization technique (e.g., transactional memory with specific consistency and progress guarantees), we can reason about suitability of a specific data structure. The analysis of the resulting concurrent implementation may give hints on how to adjust the data structure for better performance, e.g., by modifying its sequential implementation or object type.

Many questions are yet to be resolved. The concurrency of the important class of *relaxed* transactional models should be better understood [1,5,13,18]. Relaxed TMs provide weaker consistency guarantees, compared to strict serializability and opacity, such as *elastic opacity* [5]. We conjecture that, for many important applications, such as search data structures, relaxed TM implementations may allow for strictly greater concurrency than locks. Our preliminary investigations indicate that the results in Section 4.3 can be extended to show that the linked-list implementation based on $\mathcal{E}$-STM [5] (providing elastic opacity) accepts schedules that cannot be accepted by any lock-based implementations. Moreover, we conjecture that, at least with respect to (LL, set), $\mathcal{E}$-STM allows for accepting *all* schedules accepted by *HOH* and, thus, possibly by any lock-based implementation.

In the TM context, we only considered schedules triggered by committed transactions, which was enough for deriving concurrency lower bounds. One may argue that this is not sufficient if we want to prevent the user from ever observing an inconsistent state, even if its operation is not taking

effect, suggesting that strict serializability may not be an adequate TM consistency property. How the amount of concurrency is affected by extending local serializability to aborted transactions is unclear.

One more important open question is how much the amount of concurrency tells us about the actual performance of an LS-linearizable implementation on a real multiprocessor machine with relaxed memory models, given the indications that high degrees of concurrency in software TMs may have a degrading effect on the resulting performance [16].

References

[1] Y. Afek, A. Morrison, and M. Tzafrir. View transactions: Transactional model with relaxed consistency checks. In *PODC '10: Proceedings of the 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2010.

[2] M. K. Aguilera and S. Frolund. Strict linearizability and the power of aborting. Technical report, HP Laboratories Palo Alto, 2003.

[3] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* John Wiley & Sons, 2004.

[4] R. Bayer and M. Schkolnick. *Concurrency of operations on B-trees*, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[5] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC '09: Proceedings of the 23rd International Symposum on Distributed Computing*, volume 5805 of *LNCS*, pages 93–107, sep 2009.

[6] C. Ferri, A. Marongiu, B. Lipton, R. I. Bahar, T. Moreshet, L. Benini, and M. Herlihy. SoC-TM: integrated HW/SW support for transactional memory programming on embedded MPSoCs. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 39–48, 2011.

[7] W. Golab, L. Higham, and P. Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd annual ACM symposium on Theory of computing*, STOC '11, pages 373–382, 2011.

[8] V. Gramoli and R. Guerraoui. Brief announcement: transaction polymorphism. In *SPAA*, pages 311–312, 2011.

[9] V. Gramoli and R. Guerraoui. Democratizing transactional programming. In *Middleware*, volume 7049 of *LNCS*, pages 1–19. Springer, Dec 2011.

[10] V. Gramoli, D. Harmanci, and P. Felber. On the input acceptance of transactional memory. *Parallel Processing Letters*, 20(1):31–50, 2010.

[11] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory.* Morgan and Claypool, 2010.

[12] M. Herlihy. SXM: C# software transactional memory, 2005. www.cs.brown.edu/ mph/SXM/README.doc.

[13] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, New York, NY, USA, 2008. ACM.

[14] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[15] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] P. Kuznetsov and S. Ravi. On the cost of concurrency in transactional memory. In *OPODIS*, pages 112–127, 2011. full version: http://arxiv.org/abs/1103.1302.

[17] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[18] J. E. B. Moss. Open nested transactions: Semantics and support. In *WMPI*, February 2006.

[19] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[20] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC*, pages 16–25, 2010.

[21] J. Reinders. Transactional Synchronization in Haswell, 2012. http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/.

[22] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[23] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

A Sequential Implementation of Integer Set

Algorithm 1 Sequential implementation *LL* (*sorted linked list*) of *integer set* type

```
Shared variables:
   Nodes Head, Tail, X_1, ...
Local variables:
   Boolean res, ← false

Locate(s):
   prev ← Head
   curr ← prev.next
   while (curr.val < s ∨ curr = Tail) do
      prev ← curr
      curr ← curr.next
   end while
   return ⟨prev, curr⟩

Contains(s):
   ⟨prev, curr⟩ ← Locate(s)
   if curr.val = s then
      res ← true
   else
      res ← false
   return res

Insert(s):
   ⟨prev, curr⟩ ← Locate(s)
   if curr.val ≠ s then
      X_s ← new-node(s)
      X_s.next ← curr
      prev.next ← X_s
      res ← true
   else
      res ← false
   return res

Remove(s):
   ⟨prev, curr⟩ ← Locate(s)
   if curr.val = s then
      prev.next ← curr.next
      free(curr) // Deallocate node
      res ← true
   else
      res ← false
   return res
```

In this section, we recall the sequential specification of the *integer set* type (denoted *set*) and describe a sequential implementation using a *sorted linked-list* data structure.

The type *set* is specified by the following operations: *Insert*, *Remove* and *Contains*. The sequential specification of *set* is as follows: Given an set $S \subseteq \mathbb{Z}$ and an element $s \in \mathbb{Z}$:

- *Insert(s)* augments the set S with the element s if $s \notin S$ and returns *true*, otherwise S is unchanged and returns *false*
- *Remove(s)* removes the element s from S if s is not in S and returns *true*, otherwise S is unchanged and returns *false*
- *Contains(s)* returns *true* if the element s is present in S, returns *false* otherwise

The corresponding sequential implementation *LL* is presented in Algorithm 1. The implementation uses a *sorted linked list* data structure in which each node (except the *Tail*) maintains a *next* field to provide a pointer to the next node.

B Hand-over-hand locking (*HOH*)

In this section, we describe the implementation I^{HOH} (Algorithm 2) of an *integer set* type based on *hand-over-hand* locking [4]. In brief, *HOH* works as follows. Each object in the list is associated with a lock. Each node apart from the *Tail* contains a pointer *next* to a valid next node in the list when it is unlocked. An element is inserted into the list in the appropriate position while holding

the locks of the two adjacent nodes. The *Remove* node method redirects the predecessor's *next* field to make the node unreachable to other processes and then deallocates the node.

Theorem 17 (Theorem 6) *I^{HOH} is LS-linearizable with respect to (LL, set).*

Proof. Let E denote an execution of I^{HOH} and H, the corresponding history. Let $<_E$ denote an total-order on events in E and $\tilde{H}$ denote the high-level history of H. The linearization point of an operation π, denoted as ℓ_π is associated with an event performed during the lifetime of π using the following procedure. First, we obtain a completion of $\tilde{H}$ by completing every pending operation or discarding events from an operation as follows:

- Every event from an incomplete *Contains* operation is discarded.
- Every event from an incomplete *Insert* operation that has not performed the action in Line 28 of the *Insert* operation is discarded.
- Every event from an incomplete *Remove* operation that has not performed the action in Line 58 of the *Remove* operation is discarded from E; otherwise, it is completed with the step in Line 59.

Next, we obtain a sequential high-level history S by associating linearization points with operations. For every operation π, ℓ_π is defined as follows:

- If π is *Contains*, ℓ_π is associated with the successful acquisition of the lock for the last *read* on a node (excluding *Tail*) in Line 72.
- If π is *Insert*, ℓ_π is associated with Line 23 of the *Insert* operation.
- If π is *Remove*, ℓ_π is associated with Line 55 of the *Remove* operation.

High-level linearization.

Lemma 18 *If $\pi_i \rightarrow_{\tilde{H}} \pi_j$, then $\pi_i \rightarrow_S \pi_j$.*

Proof. It is easy to see that the real-time ordering of operations is respected in S since linearization points are chosen within the lifetime of the operation. □

Lemma 19 *S is consistent with the type integer set.*

Proof. If *Contains(s)* returns *true* in $\tilde{H}$, there exists *Insert(s)* that returns *true* in $\tilde{H}$ such that *Insert(s)* $\rightarrow_{\tilde{H}}$ $Contains(s)$. Then, *Insert(s)* $\rightarrow_S$ $Contains(s)$ since $\ell_{Insert(s)}$ is associated with the acquisition of the exclusive lock on node (say) X_s and no other process can read X_s until the process executing *Insert(s)* releases X_s.

Also, there exists no $\pi = Remove(s)$ that returns *true* in $\tilde{H}$ such that *Insert(s)* $\rightarrow_S \pi \rightarrow_S$ $Contains(s)$. Let processes p_1 and p_2 execute operations π and *Insert(s)* respectively. Suppose that such a π exists. Then, $\ell_{Insert(s)} <_E \ell_\pi <_E \ell_{Contains(s)}$, p_1 acquires the exclusive lock on node (say X_s) that contains the value s following which p_2 attempts to read X_s. Process p_2 can only read X_s after p_1 has released the lock on X_s. But prior to release of the lock on X_s, p_1 *physically* deletes

X_s from the list by redirecting the predecessor's *next* field to the node immediately succeeding X_s. Thus, the check in Line 78 fails and *Contains*(s) must return *false*—contradiction.

Now, if *Contains(s)* returns *false* in $\tilde{H}$, there does not exist *Insert(s)* that returns *true* such that *Insert(s)* $\rightarrow_S$ *Contains*(s) or there exists an *Insert(s)* that returns *true* and *Remove(s)* that returns *true* such that *Insert(s)* $\rightarrow_S$ *Remove(s)* $\rightarrow_S$ *Contains*(s). Again, the same arguments as above verify this claim. □

From Lemmas 19 and 18, the proof follows.

For the second part, we need to show that for all high-level operations π in H, there exists a sequential history S of LL such that $H|\pi = S|\pi$.

Consider any execution of *Contains(s)* by a process p_i in I^{HOH}. Process p_i cannot access a node deallocated from the list since the node and its predecessor are held exclusively by the process executing the *Remove* operation. Let X_s be the node that contains the value s. Suppose that p_i acquires the lock on X_s in shared mode, then process p_j executing *Remove(s)* cannot acquire the lock on X_s. Thus, *Contains(s)* precedes *Remove(s)* in the linearization. Otherwise if p_j acquires the lock in exclusive mode on X_s prior to p_i, then p_i cannot read X_s until p_j releases the lock on X_s and thus is linearized after *Remove(s)*. In either case, the execution is consistent with the sequential execution of *Contains(s)*.

Let p_i be any process executing *Insert(s)* in I^{HOH}. Then, p_i acquires the exclusive lock on node X_{s-1} (assuming the new node to be inserted is X_s). Thus, any other process that attempts to read X_{s-1} must observe the new element inserted and is linearized after the *Insert(s)*.

Let p_i be any process executing *Remove(s)* in I^{HOH}. Then, p_i acquires the exclusive lock on node X_s and X_{s-1}. Consider any concurrent process p_j that executes *Contains(s)*, p_j reads X_{s-2} and then attempts to acquire the lock on X_{s-1}, but X_{s-1} is locked by p_i. Thus, p_j waits until p_i releases the lock on X_{s-1} and then proceeds to acquire the lock on it in shared mode. Thus, *Remove(s)* is linearized prior to *Contains(s)*. Process p_j traverses the entire list until it reaches the *Tail* and returns *false*. □

C 1-progressive single-version TM implementation

Algorithm 3 describes a *1-progressive* TM implementation I^{OP}. The implementation uses a *wait-free multi-trylock* object described in [16].

Let E be any execution of the TM implemented by Algorithm 3. We assume every t-object is initialized by some fictitious committed transaction T_0 that precedes E. Let $<_E$ denote a total-order on events in E.

For a transaction T_k that appears in E, the *write set* (resp., the *read set*) of a transaction T_k, denoted *Wset*(T_k) (resp., *Rset*(T_k)), is the set of t-objects that are written (resp., read) in T_k.

Linearization points. Let H denote a linearization of $E|_{TM}$ constructed by selecting *linearization points* of tm-operations performed in $E|_{TM}$. The linearization point of a tm-operation *op*, denoted as ℓ_{op} is associated with a base object event or a tm-event performed during the lifetime of *op* using the following procedure.

First, we obtain a completion of $E|_{TM}$ by removing some pending invocations and adding responses to the remaining pending invocations involving a transaction T_k as follows:

- Every incomplete $read_k$, $write_k$ or $tryA_k$ operation is removed from $E|_{TM}$
- For every pending $tryC_k$, if some base object v_j was written (Line 32), $tryC_k$ is completed with the steps in 32 and 33 and the response C_K is added to the end of $E|_{TM}$. Otherwise, A_k is added to the end of $E|_{TM}$.

Now a linearization H of $E|_{TM}$ is obtained by associating linearization points to tm-operations in the obtained completion of $E|_{TM}$ as follows:

- For every tm-read op_k, ℓ_{op_k} is chosen as the event in Line 9 of Algorithm 3
- For every tm-write or tm-abort op_k, ℓ_{op_k} is chosen as the invocation event of op_k
- For every $op_k = tryC_k$ such that $Wset(T_k) \neq \emptyset$, ℓ_{op_k} is associated with the successful release of the lock on $Wset(T_k)$ (Line 33)
- For every $op_k = tryC_k$ such that $Wset(T_k) = \emptyset$, ℓ_{op_k} is associated with the first step performed in Line 22.

$<_H$ denotes a total-order on tm-operations in the history H.

Serialization points. The serialization of a transaction T_j, denoted as δ_{T_j} is associated with the linearization point of a tm-operation performed within the lifetime of the transaction.

We obtain a t-complete history $\bar{H}$ from H as follows:

- For every transaction T_k in H that has not invoked $tryC_k$, we insert $tryC_k \cdot A_k$ immediately after the last event of T_k in H
- For every aborted transaction T_k in H, we remove all events associated with the transaction

A serialization S is obtained by associating serialization points to transactions in $\bar{H}$ as follows:

- For every updating transaction T_k, δ_{T_k} is ℓ_{tryC_k}.
- If T_k is a read-only transaction, then ℓ_{op_k} is associated with the linearization point of the first tm-read performed in T_k

$<_S$ denotes a total-order on transactions in the t-sequential history S.

Lemma 20 *If $T_i \prec_{\bar{H}} T_j$, then $T_i <_S T_j$*

Proof. This follows from the fact that for a given transaction, its serialization point is chosen within the lifetime of the transaction implying if $T_i \prec_{\bar{H}} T_j$, then $\delta_{T_i} <_E \delta_{T_j} \implies T_i <_S T_j$ $\square$

Lemma 21 *S is legal*

Proof. Recall that S is legal if every tm-read of an object X performed by a transaction T_i returns the response of the latest value written to X in S. The latest value written to X in S is the value written by the last transaction T_j such that $T_j <_S T_i$ and $X \in Wset(T_j)$.

To prove that S is legal, we need to show that if there exists a $read_j(X)$, $X \in Rset(T_j) \cap Wset(T_i)$ that returns the value of X updated in $write_i(X, value)$ (this is well-defined since every value written

to shared-memory by a transaction is associated with the unique identifier of the transaction), then there does not exist a transaction T_k, $X \in \mathit{Wset}(T_k)$ such that $T_i <_S T_k <_S T_j$. Suppose there exists no such T_k. We need to show that $T_i <_S T_j$.

Consider two cases:

(1) Suppose that T_j is a read-only transaction. Let $\mathit{read}_j(X')$ be the first tm-read performed by T_j. By assumption, δ_{T_j} and δ_{T_i} are $\ell_{\mathit{read}_j(X')}$ and ℓ_{tryC_i} respectively. Since there exists no concurrent transaction that writes to some $X_j \in \mathit{Rset}(T_j)$, it is easy to see that T_i, T_j is a legal serialization.

(2) Suppose that T_j is an updating transaction. Thus, δ_{T_j} and δ_{T_i} are ℓ_{tryC_j} and ℓ_{tryC_i} respectively. Thus, T_i, T_j is a legal serialization.

Suppose such a T_k exists. Consider two cases:

(2) Suppose that T_j is a read-only transaction. Assume the contrary that

- there exists a $\mathit{read}_j(X)$, $X \in \mathit{Rset}(T_j) \cap \mathit{Wset}(T_i)$ that returns the value of X updated in $\mathit{write}_i(X, \mathit{value})$
- $\exists T_k$, $X \in \mathit{Wset}(T_k)$ such that $T_i <_S T_k <_S T_j$

Since $\ell_{\mathit{tryC}_i} <_E \ell_{\mathit{tryC}_k}$, T_i releases the lock on X prior to the release of the lock on X by T_k. Also, T_k can acquire the lock on X only after the release of the lock on X by T_i. By assumption, the release of the lock on X (and consequently the write of X to shared-memory) precedes the linearization of the tm-read of the first t-object $X' \in \mathit{Rset}(T_j)$ (Line 9). But then $\mathit{read}_j(X)$ cannot return the value of X written by T_i—contradiction.

(2) Suppose that T_j is an updating transaction. If there does not exist any $Y \in \mathit{Rset}(T_j) \cap \mathit{Wset}(T_k)$, $Y \neq X$ such that $\mathit{read}_j(Y)$ returns the value of Y written by T_k, then T_i, T_j, T_k is a legal serialization.

Suppose such a $Y \in \mathit{Rset}(T_j) \cap \mathit{Wset}(T_k)$ exists. Assume the contrary that $T_i <_S T_k <_S T_j$. Consider the resulting ordering of events: T_i acquires the lock on X, writes to X, releases the lock on X, T_k acquires the lock on X and Y, writes to X, Y and releases the lock on them. Since $\mathit{read}_j(X)$ returns the value of X written by T_i, $\ell_{\mathit{read}_j(X)}$ precedes the release of the lock by T_k. Subsequently, T_j performs $\mathit{read}_j(Y)$ and performs the check on Line 36 and proceeds to perform the check on Line 37. Since the timestamp of X has been updated by T_k since T_j last read it and T_k reads the value of Y written by T_k, Line 37 returns *true* and T_j returns A_j—contradiction.

□

From Lemmas 20 and 21, we have

Theorem 22 *I^{OP} is* strictly serializable*.*

Algorithm 2 Hand-over-hand locking implementation (*sorted linked list*) I^{HOH} of *integer set* type

```
1: Shared variables:
2:   Nodes Head, Tail, X_1, ...

3: Local variables:
4:   Boolean result, ℓ, ← false

5: Insert(s):
6:   prev ← Head
7:   spinLock(prev, S)
8:   curr ← prev.next
9:   spinLock(curr, S)
10:  if curr.val > s then
11:    spinLock(prev, E)
12:    goto Line 25
13:  else
14:    while (curr.val < (s − 1)∨
15:           curr = Tail) do
16:      release(prev)
17:      prev ← curr
18:      curr ← curr.next
19:      spinLock(curr, S)
20:    end while
21:    release(prev)
22:    prev ← curr
23:    spinLock(prev, E)
24:    curr ← curr.next
25:  if curr.val ≠ s then
26:    X_s ← new-node(s)
27:    X_s.next ← curr
28:    prev.next ← X_s
29:    result ← true
30:  else
31:    result ← false
32:  release(prev)
33:  release(curr)
34:  return ok

35: Remove(s):
36:  prev ← Head
37:  spinLock(prev, S)
38:  curr ← prev.next
39:  spinLock(curr, S)
40:  if curr.val ≥ s then
41:    spinLock(prev, E)
42:    spinLock(curr, E)
43:    goto Line 57
44:  else
45:    while (curr.val < (s − 1)∨
46:           curr = Tail) do
47:      release(prev)
48:      prev ← curr
49:      curr ← curr.next
50:      spinLock(curr, S)
51:    end while
52:    release(prev)
53:    prev ← curr
54:    curr ← curr.next
55:    spinLock(curr, E)
56:    spinLock(prev, E)
57:  if curr.val = s then
58:    prev.next ← curr.next
59:    free(curr)
60:    result ← true
61:  else
62:    result ← false
63:  release(prev)
64:  release(curr)
65:  return ok

66: Contains(s):
67:  prev ← Head
68:  spinLock(prev, S)
69:  curr ← prev.next
70:  spinLock(curr, S)
71:  while (curr.val < s∨
72:         curr = Tail) do
73:    release(prev)
74:    prev ← curr
75:    curr ← curr.next
76:    spinLock(curr, S)
77:  end while
78:  if curr.val = s then
79:    result ← true
80:  else
81:    result ← false
82:  release(prev)
83:  release(curr)
84:  return result

85: spinLock(X, mode):
86:  ℓ ← false
87:  while ℓ do
88:    ℓ ← acquire^mode(X)
89:  end while
```

Algorithm 3 I^{OP};Implementation of T_k executed by process p_i

Shared variables:
  $clock \in \mathbb{N}$, initially 0 // Global clock
  v_j, for each t-object X_j
  $v_j.ts$, a timestamp for each object, initially 0
  $v_j.tid$, id of transaction that writes to v_j, initially T_0
  $v_j.val$, value of v_j
  L, multi-trylock object

read$_k(X_j)$**:**
  $ov_j := read(v_j)$
  if $X_j \notin Rset(T_k)$ **then**
    $Rset(T_k) := Rset(T_k) \cup \{X_j\}$
  return $ov_j.val$

write$_k(X_j, v)$**:**
  $nv_j.val := v$
  if $X_j \notin Wset(T_k)$ **then**
    $Wset(T_k) := Wset(T_k) \cup \{X_j\}$
  return ok

tryA$_k$**():**
  return A_k

tryC$_k$**():**
  if $|Wset(T_k)| = \emptyset$ **then**
    if isInvalid1() **then**
      return A_k
  locked := $L.acquire(Wset(T_k))$
  if not locked **then**
    return A_k
  if isInvalid2() **then**
    $L.release(Wset(T_k))$
    return A_k
  time:=clock() // Every t-object in *Wset* is assigned this timestamp
  for all $X_j \in Wset(T_k)$ **do**
    $write(\{v_j.ts, v_j.tid, v_j.val\}, \{time, k, nv_j.val\})$
  $L.release(Wset(T_k))$
  return C_k

Function: isInvalid1**():**
  if $\exists X_i \in Rset(T_k)$:$ov_i.ts < v_i.ts$ **then**
    if $\exists X_j \in Rset(T_k)$:$(v_i.ts < ov_j.ts) \wedge (ov_i.ts < ov_j.ts)$ **then**
      return *true*
  return *false*

Function: isInvalid2**():**
  if $\exists X_j \in Wset(T_k) : L.isContended(X_j)$ **then**
    return *true*
  if $\exists X_i \in Rset(T_k)$:$(ov_i.ts < v_i.ts)$ **then**
    return *true*
  return *false*